Étape 14 : Finalisation des combats.

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.0

But

Simuler les mécanismes de prédation et de combat.

Description

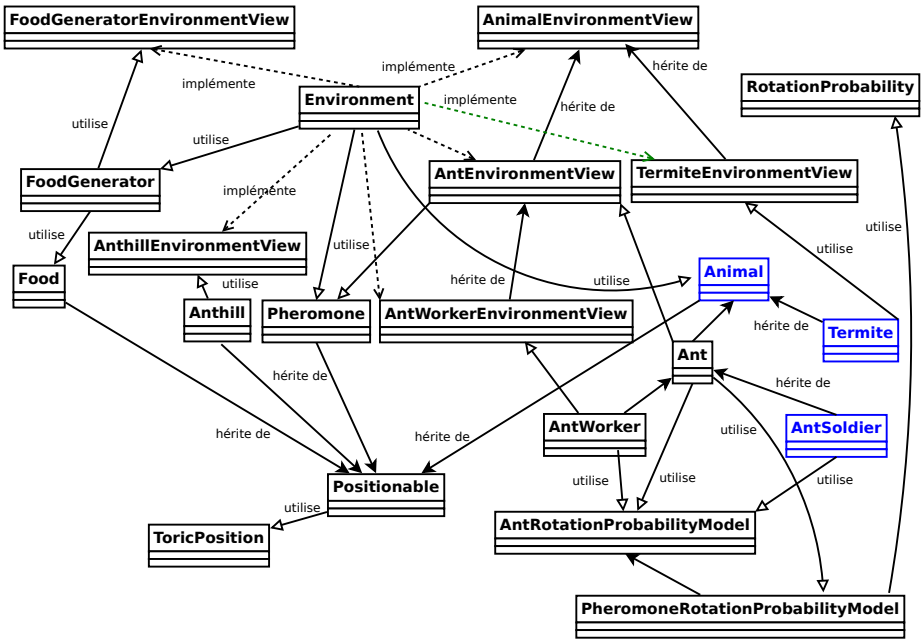

Figure 1: Composants impliqués dans l'étape 14 (en noir les composants déjà codés, en vert les nouveaux composants, en bleu ceux retouchés)

Il s'agit dans cette ultime étape de simuler les combats entres animaux ennemis. L'introduction de ce mécanisme supplémentaire nécessite d'enrichir la façon de

gérer le comportement des animaux, notamment en le faisant dépendre d'un *état* (un animal en train de combattre ne se comporte pas selon les mêmes mécanismes qu'un animal se déplaçant sans souci d'un ennemi par exemple). Il s'agira ensuite pour l'essentiel de finaliser la modélisation du comportement spécifique des prédateurs (fourmis soldates et termites).

La mise en place de ces nouveautés implique les composants illustrés dans la figure 1.

Implémentation

Le comportement des animaux dépend de leur état

Nous souhaitons que les animaux puissent engager des combats et revenir à leur comportement habituel une fois les confrontations terminées (s'ils y survivent). Il faut donc pouvoir désormais modéliser différents *états* de l'animal qui conditionneront son mode d'action :

- un état `ATTACK` signifiant qu'il est engagé dans un combat (et dans ce cas ne se déplace plus) ;
- un état `IDLE` signifiant qu'il n'est pas engagé dans un combat : l'animal aura alors un comportement spécifique à son type (pour une fourmi ouvrière, cela correspondra à chercher les sources de nourriture) ;
- un état `ESCAPING` signifiant qu'il s'évade d'un combat, et se déplace au hasard, sans attaquer quoi que ce soit, jusqu'à ce qu'il soit hors de portée de tous les ennemis. Ensuite, l'animal passe à l'état `IDLE`.

Le plus simple pour modéliser les états `ATTACK`, `IDLE` et `ESCAPING` en Java est de créer une énumération. Vous placerez l'énumération `State` au sein de la classe `Animal` et elle devra être publique (en tant que type `enum` ; l'attribut correspondant sera, bien sûr, privé).

Il vous est également demandé d'écrire les deux méthodes suivantes :

- `State getState()` retournant l'état de l'animal ;
- `void setState(State newState)` modifiant l'état de l'animal.

Vous compléterez enfin la méthode `toString()` de sorte à ce qu'elle permette l'affichache de l'état. Elle retournera une représentation textuelle d'un `Animal` selon le format de l'exemple suivant :

```
Position : 20.0, 30.0
Speed : 120.0
HitPoints : 500
LifeSpan : 30.000000
State : IDLE
```

Modification de `Animal.update`

L'algorithme permettant de faire évoluer les animaux se contentait d'en assurer le déplacement (avec ou sans utilisation de la phéromone) et de gérer la durée de vie. Il s'agit maintenant de l'étendre de sorte à y incorporer la simulation de combats.

Concrètement, nous devrons modifier la méthode `update` d'`Animal` pour prendre en compte les trois états que nous venons d'ajouter. L'idée est la suivante :

- Si l'animal est dans l'état `ATTACK` :
  - Si l'animal est en mesure d'attaquer, il appellera la méthode `fight` (voir ci-après)
  - Sinon, il bascule en mode `ESCAPING` et remet à zéro son `attackDuration`.
- Sinon si l'animal est dans l'état `ESCAPING`, il appellera simplement la méthode `escape` (voir ci-après)
- Sinon, il appellera la méthode `specificBehaviorDispatch`.

Combats

Nous allons ici écrire les méthodes mentionnées auparavant, à savoir `fight` et `escape`. Avant cela, nous allons écrire une méthode (publique) `boolean canAttack()` qui renvoie `true` si l'animal est en mesure d'attaquer.

On considère qu'un animal est capable d'attaquer s'il n'est pas en mode `ESCAPING` et si `attackDuration` est inférieure ou égale à `getMaxAttackDuration()`. Pour déterminer si un temps `t1` est inférieur ou égal à un temps `t2`, vous pouvez utiliser `t1.compareTo(t2) <= 0`.

Pour `escape`, sa signature sera

```
void escape(AnimalEnvironmentView env, Time dt)
```

Cette méthode est relativement simple : elle se contente d'appeler `move` ; de plus si l'animal n'est plus visible des ennemis, l'animal rebascule en mode `IDLE`.

Enfin, la méthode

```
void fight(AnimalEnvironmentView env, Time dt)
```

s'occupera du combat à proprement parler. L'algorithme est le suivant :

- On choisit l'ennemi visible le plus proche
  - S'il y en a un
    - Changer le mode de l'ennemi en `ATTACK`, ainsi que celui de `this` (si ce n'est pas déjà le cas)

    - Infliger des dégâts à l'ennemi compris entre `getMinAttackStrength()` et `getMaxAttackStrength()`.
    - Ajouter `dt` à `attackDuration`.
  - Sinon, on remet à zéro son `attackDuration` et s'il est en mode `ATTACK`, il bascule en mode `ESCAPING`.

Notes

- Les dégâts infligés à l'ennemi devront être uniformément réparti entre `getMinAttackStrength()` et `getMaxAttackStrength()`. Vous pourrez utiliser pour cela la méthode statique `double getValue(double min, double max)` de la classe `UniformDistribution`.
- Souvenez-vous qu'il faut écrire `attackDuration = attackDuration.plus(dt)` et pas simplement `attackDuration.plus(dt)` pour "ajouter" `dt` à `attackDuration`, car la méthode `plus` renvoie une *nouvelle* instance de `Time`.
- Pour réinitialiser `attackDuration`, vous pouvez écrire `attackDuration = Time.ZERO`.
- Vous pouvez utiliser la méthode statique `closestFromPoint` de la classe `Utils` (en passant `this` et la liste des ennemis comme arguments) afin d'obtenir l'ennemi le plus proche.

Fourmis soldates & termites

Jusqu'à maintenant, les fourmis soldates et les termites se contentaient de se déplacer, sans avoir un comportement vraiment intéressant.

Il ne nous reste plus maintenant qu'à ajouter l'appel à `fight` dans leur méthode de comportement spécifique (`seekForEnemies`). Cela leur permettra de passer activement en mode `ATTACK` (en effet, dans `Animal`, la méthode `update` appelle `fight` seulement lorsque l'animal est en combat, mais à aucun moment `update` ne change directement l'état de l'animal à `ATTACK` ; elle gère uniquement l'auto-défense de l'animal).

Tests et soumission

Tests locaux

Nous vous encourageons, comme toujours à continuer d'ajouter vos propres tests ponctuels dans le fichier `ch/epfl/moocprog/tests/Main.java`.

Pour les tests graphiques, vous pouvez commencer par utiliser le fichier de configuration `config-step13.cfg` suggéré à l'étape précédente (celui où

l'environnement est infesté de termites). La différence avec l'étape précédente est que vous devriez observer régulièrement des termites et fourmis arrêter de se déplacer pour combattre. Les combats se soldent par la disparition des ouvrières. Les soldates sortent quant à elles vainqueures du combat de façon aléatoire.

Vous pouvez jouer sur les paramètres du panneau de droite pour renforcer la capacité de survie des fourmis et diminuer la force de combat des termites (par exemple en réduisant la force d'attaque des termites à 1 et en réduisant leur durée maximale de combat à 0.5, vous devriez pouvoir observer des ouvrières sortant vivantes d'un combat).

Pour observer uniquement le rapport entre termites et soldates vous pouvez configurer les fourmilières de sorte à ce qu'elle ne produisent que des soldates (touche `Restart` nécessaire pour la prise en compte de ce paramètre). Vous disposez aussi du bouton `hitPoints` pour observer la perte des points de vie pendant un combat.

Vous pouvez ensuite dans la même veine, créer un fichier `config-step14.cfg` d'abord avec une seule fourmilière et sans termites pour vérifier qu'aucun combat ne s'engage entre fourmis de la même fourmilière. Vous pouvez ensuite y ajouter une seconde fourmilière (toujours sans termites) pour observer des combats entre fourmis de fourmilières rivales.

Avec le fichier fourni `config.cfg`, l'ensemble des comportements souhaités dans le projet devrait être observables.

Des vidéos donnant des exemples d'exécution possibles sont fournies dans le matériel de la semaine 14.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **tous** vos fichiers sources personnels, depuis la racine `ch/` ; ceux de cette étape, mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

```
ch/epfl/moocprog/AnimalEnvironmentView.java
ch/epfl/moocprog/Animal.java
ch/epfl/moocprog/AntEnvironmentView.java
ch/epfl/moocprog/AnthillEnvironmentView.java
ch/epfl/moocprog/Anthill.java
ch/epfl/moocprog/Ant.java
ch/epfl/moocprog/AntRotationProbabilityModel.java
ch/epfl/moocprog/AntSoldier.java
ch/epfl/moocprog/AntWorkerEnvironmentView.java
ch/epfl/moocprog/AntWorker.java
ch/epfl/moocprog/Environment.java
ch/epfl/moocprog/FoodGeneratorEnvironmentView.java
```

`ch/epfl/moocprog/FoodGenerator.java`
`ch/epfl/moocprog/Food.java`
`ch/epfl/moocprog/Pheromone.java`
`ch/epfl/moocprog/PheromoneRotationProbabilityModel.java`
`ch/epfl/moocprog/Positionable.java`
`ch/epfl/moocprog/RotationProbability.java`
`ch/epfl/moocprog/TermiteEnvironmentView.java`
`ch/epfl/moocprog/Termite.java`
`ch/epfl/moocprog/ToricPosition.java`

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres).